UNITED STATES PATENT APPLICATION

FOR

SYSTEMS AND METHODS FOR MULTI-VIEW DEBUGGING ENVIRONMENT

Inventors:

Josh Eckels
William Pugh

CERTIFICATE OF MAILING BY "EXPRESS MAIL"
UNDER 37 C.F.R. § 1.10

"Express Mail" mailing label number: **EV 386447533 US**
Date of Mailing: 2/23, 2004

I hereby certify that this correspondence is being deposited with the United States Postal Service, utilizing the "Express Mail Post Office to Addressee" service addressed to **Mail Stop PATENT APPLICATION, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450**, and mailed on the above Date of Mailing with the above "Express Mail" mailing label number.

Tina M. Galdos
Signature Date: 2/23, 2004

Systems and Methods for Multi-View Debugging Environment

Inventors:
Josh Eckels
William Pugh

COPYRIGHT NOTICE

CLAIM OF PRIORITY

[0002] This application claims priority from the following application, which is hereby incorporated by reference in its entirety:

[0003] U.S. Provisional Patent Application No. 60/451,368, entitled "Systems and Methods for Multi-View Debugging Environment" by Josh Eckels and William Pugh, filed March 1, 2003 (Attorney Docket No. BEAS-1436US1).

FIELD OF THE INVENTION

[0004] The present invention relates to the debugging of an executing software program.

BACKGROUND

[0005] Many software programs contain complex data structures, and software developers rely on predefined libraries data structures (e.g. classes), or create their own libraries. In these cases, when a developer encounters a debugging problem in executing a software program, they are sometimes more interested in investigating the abstract contents of the data structures, i.e., the attributes of their interests during the execution of the software program, rather than the physical structures used to represent the abstract contents. Unfortunately, existing debugging systems or "debuggers" often present developers with internal details of these structures, which makes it difficult to determine the abstract contents of actual interest. For example, the developer might use a data

structure called a List to represent an ordered collection of items on an invoice. In the debugger, the developer might wish to see the list of items and their attributes (e.g., quantity, price, description). However, internally the List data structure is implemented as a linked list of nodes. Therefore, in order to understand or monitor the contents of the List using a prior art debugger, the developer has to follow a long series of pointers between nodes and examine variables names along the way, such as nodeptr and nextptr that have little to do with the list of invoice items the developer wants to monitor.

[0006] In some cases, this extraneous information can include details of the data structures created by the supplier of a library, and for which the developer may have poor facilities for understanding. Even if the developer understands the objects and structures, the problem they are addressing may be focused on the abstract content represented by the data structure, rather than the physical structure.

[0007] In addition to these problems, some existing software debuggers provide little or no control over the format for presentation of the information and have little or no editing capability for data values within objects or data structures. Thus, in many cases, the software developer is presented with unwanted information, rigidly formatted, and with limited or no editing capability.

[0008] In addition to the difficulties discussed above, debugging of Java Server Pages (JSP pages) presents a unique set of difficulties. These difficulties arise from a number of sources inherent in the way JSP pages are developed and deployed. In many cases, the code executed to implement the JSP page is "machine generated" and is embedded into a servlet running in a complex runtime environment. Thus, it may be very difficult for a developer to understand the mapping between the JSP source code they have created and wish to debug, and the machine generated code they would see through an existing debugger applied to the runtime environment. Further complicating this situation is the common use of tags that redirect the execution path within the JSP page. Also, JSP pages also consume and produce streams of data.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] **Figure 1** is a diagram showing a software debugging system that can be used in accordance with one embodiment of the present invention.

[0010] **Figure 2** is an exemplary GUI for debugging of a JSP page in accordance with one embodiment of the invention.

[0011] **Figure 3** is an exemplary GUI for debugging of a JSP page in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

[0012] The invention is illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an" or "one" or "some" embodiment(s) in this disclosure are not necessarily to the same embodiment, and such references mean at least one.

[0013] **Figure 1** illustrates systems and methods in accordance with some embodiments of the present invention that can provide software developers with a software debugging environment. Although this diagram depicts objects/processes as logically separate, such depiction is merely for illustrative purposes. It will be apparent to those skilled in the art that the objects/processes portrayed in this figure can be arbitrarily combined or divided into separate software, firmware and/or hardware components. Furthermore, it will also be apparent to those skilled in the art that such objects/processes, regardless of how they are combined or divided, can execute on the same computing device or can be distributed among different computing devices connected by one or more networks or other suitable communication means.

[0014] Referring to **Figure 1**, the software debugging environment **100** is capable of utilizing multiple viewing capabilities to provide debugging flexibility when executing a software program **101** having one or more data structures. Developers can select between one or more abstract views **107, 108,** and **109,** which are views displaying only the abstract contents of interest to the developer during the execution of the software program, in order to see different abstract content(s) **102** of one or more data structures of the software program being executed. Each abstract view can have a filter **103,104, or 105**, which has specific properties determining which contents of a data structure are displayed, the format in which they are displayed, and how the underlying data structure can be modified through the abstract view. In some embodiments, the selection of views, filters defining the views, and the editing of the contents of the data structures shown in the views can be set interactively. In some cases, this interaction can occur through an interface **113** of an Integrated Development Environment (IDE) **114** containing the debugger. In other cases, the filters of views in the debugging environment can be determined by configuration information supplied in a file **106**, and perhaps by the context.

In some embodiments, two or more views of the same content(s) of a data structure can be simultaneously displayed for the developer.

[0015] In one embodiment, a developer can view the abstract contents of a data structure, rather than viewing the actual internal details of how the contents are stored in memory. A filter may extract the contents of interest from the data structure and format them for display to the user. For example, a Java language developer may not care about the internal data structures used to implement an array list such as java.util.ArrayList, but would like to view the abstract contents such as the items stored in the list. As another example, a debugger can make extensive use of a viewing mechanism for a developer examining the contents contained in a Java Servlet Page (JSP), without the need to see the surrounding structure or the generated servlet code.

[0016] In some embodiments, a view can also include customized editors **110, 111,** and **112**, allowing the user to set values for the displayed abstract contents of a data structure through an abstract view. In some embodiments, the editor may perform validation of the user's input against allowed values for the contents. In some embodiments the user, possibly in the IDE environment, can set the properties of the editor interactively via a filter. In some embodiments, the properties of the editor are defined in configuration files. In some embodiments, the editors may provide the capability set multiple underlying contents in a single operation. As an example of an editor in a customized view, an editor for java.awt.Color can provide a color selection dialog as its editor.

[0017] In some embodiments, a user can select various views to display by toggling each one of them on and off individually. The ability to toggle the views can be useful if a user wishes to view a content's underlying data structure, for example. As another example, a user can select multiple views to be displayed simultaneously. As an example, the user may be able to right-click on a node in the local content windows or watch windows and set it to use any of the one or more views that match for its content's type. Ideally, there will be at least one match for that type.

[0018] In some embodiments, filtering is performed in an IDE. The filters can extract, and in some instances format, the information to be displayed in a particular abstract view. In some embodiments the filters are arbitrarily configurable (e.g. they may be defined in a language, such as an XML). In some cases, a configuration file can be used to define the properties of the filter. In other cases the user may set the properties of the filter, and may

do so interactively and/or using the facilities of the IDE. In some cases filters can be applied to data in a runtime-messaging environment.

[0019] As an example, a developer may wish to look at the contents of a customer order. The developer may not need to see how the order is constructed, such as whether the items in the order are contained within a linked list of data structures. Filters can be defined to extract the local contents of interest (i.e. item name, item price, quantity) and display those contents in one or more convenient formats. For some contents the developer may only wish to see the character representation. For other contents, the developer may only wish to see decimal numeric representations, or an integer and hexadecimal representation (2 views) may be desired. A debugger according to some embodiments in the present invention can apply the appropriate filters to extract the desired contents and transformation those contents to the desired display formats. The views of these contents can be displayed in the IDE. If a developer wishes to change a view to see another representation of some contents, eliminate contents from the view, or add contents to a view, the developer can do so either interactively, such as through the IDE or by changing a configuration file.

[0020] In some embodiments, one or more special facilities can be included to support the debugging of a JSP page and the machine generated servlets that implement it. These faculties can be applied in a coordinated manner, along with capabilities already described to create a complete environment for efficiently debugging JSP pages. These facilities can include, for example:

- use of special purpose filters to extract, display code and values of contents of interest, and map the code and values to the formats used in a source code in a JSP page, for use with executing JSP servlets;
- the ability to follow execution paths through several levels of redirection; and
- the use of specialized filters to extract data from and manipulate the contents of buffers used to transmit and receive streaming data.

[0020] In some embodiments, tags can be used to redirect the execution path within a JSP page. An effective debugger can follow this flow of execution correctly, especially in cases where a developer is interested in debugging the code in the redirection path. As an example, JSP developers typically use one or more tag libraries, which can be standard libraries or specially developed libraries, to define functionality without the need to reproduce large amounts of code in line.

[0021] In one embodiment, JSP pages can consume and produce streams of data. In such cases it can be necessary for an effective debugger to provide access into buffers, for completed and in-progress streams, to allow the developer to view, and possibly manipulate, contents. As an example, a JSP may use one or more buffers to receive and transmit HTTP streams during execution.

[0022] The follow discussion provides exemplary implementation of abstract views for debugging that can be used in accordance with various embodiments. Throughout this discussion, an example is developed using the Java programming language. It should be understood that the advantages obtained through these enablements can be obtained using other programming languages as known and used in the art. Further, it should be understood that a wider range of views can be created and defined, in Java or another programming language, and that the scope of the invention is not defined by these examples.

[0023] A view for byte, short, char, int, and long can present a user with the option for two views each, one for showing the value in hex and one for showing the value in decimal. Decimal may be the default. In some cases, the editors may not allow non-numeric data or accept values that are beyond the minimum or maximum values for each type.

[0024] A view for char can present the user with the choice of views showing the character value, a Unicode escape syntax value, and the hex value. The default view may show a single quoted character. The editor may not allow the user to enter more than one character or a valid Unicode escape value.

[0025] A view for float and double can present the user with a single view for these two types that shows the default string representation of their types (including "NaN", defined in IEEE floating point number standards, etc). The editor may only allow the user to enter a value that will parse as a float or double. In some cases alternative views (e.g. showing hex values) may be available.

[0026] A view for Boolean can display "true" or "false", or alternatively, "1" and "0". The editor may only allow the two valid values as input.

[0027] A view for array can show a selected range of array elements as children. Users may be able to easily select what elements are to be shown. The node corresponding to the array itself should display the type of the array and its length. Editing support may correspond to the elemental types in the array. In some embodiments, no editing is provided in the view.

[0028] For developers using the Java programming language, one or more views for java.util.List can provide access to the contents of list objects. This view may show the list as an array of the same length, letting the user choose the range of indices to display. Editing support may correspond to the elemental types in the array. In some embodiments, no editing is provided in the view.

[0029] For developers using the Java programming language, one or more views for java.util.Set can provide access to the contents of set objects. The view can show the set as an array of the same length, letting the user choose the range of indices to display. Editing support may correspond to the elemental types in the array. In some embodiments, no editing may be provided in the view.

[0030] For developers using the Java programming language, one or more views for java.util.Map may provide access to the contents of map objects. These views may show the key-value pairs as an array of the same length. Those nodes can be expanded and contain two children, the key and the value objects. Editing support may correspond to the elemental types in the array, and may place restrictions on key values. In some embodiments, no editing is provided in the view.

[0031] For developers using the Java programming language, one or more views for java.util.Date and java.util.GregorianCalendar may provide display the underlying date and time in these objects. The editor may use a SimpleDateFormat to validate the value before passing it into a date object.

[0032] For developers using the Java programming language, one or more views for java.lang.StringBuffer may provide access to the current length, current capacity, and string contents. Editing support may be supplied for the elemental character values, or not at all.

[0033] For developers using the Java programming language, one or more views for java.awt.* and javax.swing.* classes can provide access to the contents of these data structures. For example, Containers might have an easy way to see their children without having to dive into the internal implementations. Other views might be simpler, like the java.awt.Color example above, or a summary view of java.awt.Dimension that shows its height and width at the root node instead of making the user drill down into its children.

[0034] In some embodiments, views can be at least partly defined using an XML-based language. A number of definition parameters can be defined for views. Some embodiments may use additional parameters, or eliminate some of the parameters discussed. Some possible examples include:

- A "priority" parameter may be assigned to the view. For values that match more than one view, the view with the highest priority can be chosen by default. All of the matching views can be available to the user on a right-click, sorted by priority.
- A "value type" parameter defines type displayed in the view. For example, a value type can be either the string that the debugger will report for the value's type or "*". The latter indicates the view is valid for all expression values.
- A "view class" parameter defines the name of a class that implements in the view. In some embodiments using the Java language this may be implemented as a class, such as com.bea.ide.debug.IDebugExpressionView.
- A "description" parameter may be a string that will be shown to describe the view after right-clicking on an expression in the locals or watch windows.
- In some embodiments an "invalidates default" attribute may be set to true, if the view matches an expression's type exactly. Any views with a valueType of "*" will not be included in the list of matching views for a given type. This capability is used when there is no need in the default view for primitive types, which which will generally have client-side validation.

[0035] As an example, a debugger view definition can use a construct of a form such as:

```
<extension-xml id="urn:com-bea-ide:debugExpressionViews">
  <view priority="PRI" valueType="VALUE_TYPE" class="VIEW_CLASS"
        invalidatesDefault="true" description="DESCRIPTION" />
</extension-xml>
```

[0036] Each debugger view can address the possibility that the contents being accessed may not be in an internally consistent state. For example, if an implementation of java.util.List has an int member variable for its length and an array member variable to hold its elements, the debugger view may have to display the list in the middle of an add operation when the length member variable is 11 but the array member variable is only 10 elements in length.

[0037] In some embodiments, views can be implemented using method calls. Such implementations may need to use techniques to prevent potential deadlocks that can arise, for example, since Java Debug Interface (JDI)'s method invocation respects object monitor locking in a Java language environment. To avoid these potential deadlocks, the view may require knowledge of the implementation of a class, even when it could retrieve all its needed data through method calls. For example, using method calls a single view could be used for all java.util.Lists, but a view may require knowledge of the internal

structure of java.util.ArrayList, java.util.LinkedList, etc, if method calls are not used. Safeguards may be implemented to prevent any changes to the implementation of those classes causing failures while executing one or more of the views.

[0038] In some embodiments, the names used to identify modules can reflect the names used for the source files, to prevent user confusion. As an example, when debugging a JSP, the views for debugging should show the name of the source .jsp file instead of the generated servlet .java file. The user may remain fairly ignorant of the generated servlet file, and the names used, altogether.

[0039] Some embodiments can apply one or more specific filters to viewing and manipulating values used in JSP pages and the servlets generated to implement them. In general, these filters can be intended to hide contents internal to the runtime environment, and just present the user with the information they would be able to get programmatically from within a JSP page. The user can toggle between the filtered view and the full view, which can include, for example, a default Java object view.

[0040] Some examples of suitable filters are shown below. In some cases, these examples assume the Java programming language is used for the JSP page or machine generated servlet code and the HTTP protocol is used for streaming data communications.

- URL information, including port, query string, etc.
- HTTP method, including post, get, etc.
- HTTP headers, including name, value, etc.
- Parameters, including name, value, etc.
- Attributes, including name, value, etc.
- Cookies, including name, domain, max age, value, etc.
- Session information, including session ID, values, session length, etc.

[0041] Some embodiments can contain facilities to accommodate the use of tags and tag libraries. One or more filters can be used to extract and display information of interest when tags and tag libraries are encountered. This capability can include the ability to follow (or not follow as required) the execution of the program through one or more redirections and can allow the user to view the current streams, relative to the tag library. This information can be useful in many situations including, with nested tags, the output from one tag library can become the input for a different tag library and where the intermediate data is never sent to the final output stream. Some embodiments can provide the user with control over which tag libraries are viewed and which are skipped.

[0042] Some embodiments can include the capability to examine the contents of data streams. Contents of the buffers and information concerning the buffers can be extracted and displayed using one or more filters. Local watch windows can use debugger views to display some summary information about the streams (for example, the number of bytes that have been written to the stream, how many bytes are still in the buffer, etc). A user can click for a dialog with the full contents of the stream in addition to the same summary information. The user may be able to toggle between text and binary views of the streams, or show both views at the same time. For commonly used streams, like the output of JSP to the client, there can be an additional debugger window. In some embodiments, the view of the stream, and possibly the filters applied, may differentiate between flushed and buffered bytes.

[0043] Some embodiments can extract the streaming data by inserting a wrapper or "writer" class around the JSP servlet. For example, a wrapper class may be placed around a JSP in the runtime environment. The wrapper or "writer" can capture all of the stream output or input as it is written, and may also control clearing the buffer. In some embodiments, the IDE may detect local contents whose values are objects for the wrapper class and present them to the user as a list of streams. The user can select a stream, and the IDE will inspect though the wrapper to determine the contents of the stream and possibly determine how much contents have already been flushed.

[0044] Some embodiments can allow the definition of breakpoints, such as for JSP page debugging. These breakpoints can be defined for the user written code of the JSP page, the tags or the tag library or the machine generated servlet code. For example, in a JSP, breakpoints may be defined for lines with Java code, JSP tags, or tag library calls. In some embodiments, breakpoints set on lines without a defined breakpoint type may be ignored.

[0045] Stepping though the execution of a page such as a JSP page can be quite different from stepping though ordinary code. In many cases the developer may not wish to see the actual machine generated code of the servlet stepped through, but rather something that resembles the source code they have written. In some cases, a developer may wish to examine code within tag libraries, such as when the developer is developing a tag library, or attempting to find a bug in someone else's tag library. In other cases, a developer may not want to see the contents of a tag library, such as when a predefined tag library is being used. For example, these developers may view tag libraries in a similar

way to the way Java developers view the Java collections classes – something to be used but not debugged.

[0046] In some embodiments, users can set method breakpoints within specific tag libraries. The debugger and IDE may inspect the JSP page and possibly deployment descriptors to determine the classes and methods implemented within the tag library. Generally, only methods specified by the tag library interfaces will be shown, and the list may be filtered further to only include methods for which source code is available. In some embodiments, these results can be displayed in the IDE and the user may be able to right click on a tag library, possibly using a source view, to access this functionality. The breakpoints created can show up as normal method breakpoints in both the breakpoint window and the source editor.

[0047] In some embodiments, stepping out of a tag library method, or stepping over off the end of the method, will cause the debugger to continue on to the next tag library method that has been enabled as a step into location. If the last location has been passed, the debugger can step to the next action in the JSP page. Between the tag library methods, the debugger may step back to the JSP source. In some cases the user may not be able to tell which method call just completed or which call will be next, by examining the JSP page source. However, in these cases, the user can use the tag library breakpoints functionality to control which tag library methods get hit.

[0048] The following table shows some possible examples of breakpoints for JSP pages using the Java programming language and HTTP transport protocol:

Current location	**Action**	**Result**
Call in a JSP to a tag library	Step in	Steps into the next call to the tag library
	Step over	Steps to next JSP action
	Step out	Finishes the JSP request
Tag library code	Step in	Normal Java step in
	Step over	Normal Java step over, stepping off the end of the tag library method steps back to the tag library call in the JSP
	Step out	Steps back to the tag library call in the JSP
Java code snippet	Step in	Performs a normal Java step in, stepping in to the method call if present

	Step over	Performs a normal Java step over, stepping to the next line of Java code in the snippet or to the next JSP action
	Step out	Finishes the JSP request
"<%= value %>" syntax	Step in	Steps to the next JSP action
	Step over	Steps to the next JSP action
	Step out	Finishes the JSP request
Unescaped HTML	N/A	Not breakpointable or stoppable

Table 1: Exemplary Breakpoints for JSP Pages in Some Embodiments

[0049] In some embodiments, a JSP debugger can provide a user interface, possibly as part of the IDE. Such IDE can be implemented on any Java-based application platform, such as the commonly used WebLogic developed by Bea Systems. This user interface may include some of the functionality discussed here. The specific examples in this section are for use with JSP pages using the Java programming language.

[0050] Some embodiments can include a dialog, such as that shown in **Figure 2**, to allow a user to select which method(s) in the tag library should be hit when stepping into a tag library tag from a JSP. The dialog may only let the user select the tag library methods for which the IDE can find source code. It may not show tag library methods that a super class implements, but for which the IDE couldn't find the source code. This may be a common case with Java, for example, since many tag library implementations will extend TagSupport, and most users won't have the source for the JSP API. In some embodiments, this dialog may also show the implementing class' name, as well as the tag that refers to that class (for example, <prefix:tagName />), possibly as a header.

[0051] Other useful information displayed can include the class that actually implements the method (since a tag library may not override its superclass's implementation), and the tag library interface that specifies the method. This can be done as a tooltip for each of the checkboxes.

[0052] Some embodiments can include one or more stream windows as shown in **Figure 3** to allow a user to inspect the contents of streams. Within such a window, the contents of the stream can be color-coded to reflect its status. The codes may help to differentiate between, for example:

- Flushed or not flushed
- New output since last step or breakpoint and preexisting output

Some embodiments may include a color-coding key within the window.

[0053] In some embodiments, a JSP debugger can provide information to the JSP compiler for use when creating the JSP servlet. This can include information to create required interfaces and wrapper classes as required for the operation of the debugger. For example debugger specific information may need to be included in a JSP .class files to facilitate debugging.

[0054] In some embodiments, line numbers in the class file displayed by the JSP debugger can point to the source code file for the JSP. In some cases, only lines with user code (either tag library calls or programming language snippets) may be marked in the class file. As an example, lines in a JSP that just contain HTML may not have line numbers marked in the class file, since these are not code in the programming language. Such an embodiment can make the stepping behavior seem more natural to the developer.

[0055] In some embodiments, a source file name displayed and used in the class file is the file name of the programming language source code file for the JSP. The path used can be relative to the root of the project in the IDE.

[0056] In some embodiments, a compiler may only insert code for using the wrapper class ("writer" class) if debug symbols are turned on. Once this has been done, the servlet can be generated. At run-time, a generated servlet can call an invoke method, which may be a static method. This method may return a result indicating if the IDE is currently in debugging mode. If so, the wrapper class can be inserted at this time. If not, the server will remain unwrapped in the runtime environment.

[0057] In some embodiments a wrapper class can be defined by one or more abstract methods. In some cases, specific methods used for debugging operations on streams are implemented. For example a method that returns all of the bytes that have been written to the buffer, flushed and un-flushed may be used. Additional methods may be needed if it's too slow to fetch the whole page's content each time, and possibly for determining which contents are new.

[0058] In some embodiments, a debugger interface can be employed for view control and interaction. Such an interface can define the methods that the debugger uses to determine how to display the expression, and how to edit the expression's value. In some cases the debugger can determine the list of field implementations by a look up, possibly through the proxy. The look-up may provide the appropriate filers and may assign values to parameters for the view. The debugger interface com.bea.ide.debug.IDebugExpression

view, for example, can be implemented by classes wishing to provide a particular abstract view of a physical structure using the Java programming language.

[0059] One embodiment may be implemented using a conventional general purpose or a specialized digital computer or microprocessor(s) programmed according to the teachings of the present disclosure, as will be apparent to those skilled in the computer art. Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art. The invention may also be implemented by the preparation of integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be readily apparent to those skilled in the art.

[0060] One embodiment includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the features presented herein. The storage medium can include, but is not limited to, any type of disk including floppy disks, optical discs, DVD, CD-ROMs, micro drive, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, DRAMs, VRAMs, flash memory devices, magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

[0061] Stored on any one of the computer readable medium (media), the present invention includes software for controlling both the hardware of the general purpose/specialized computer or microprocessor, and for enabling the computer or microprocessor to interact with a human user or other mechanism utilizing the results of the present invention. Such software may include, but is not limited to, device drivers, operating systems, execution environments/containers, and applications.

[0062] The foregoing description of the preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in the art. Particularly, while the concept "content" is used for in the embodiments of the systems and methods described above, it will be evident that such concept can be interchangeably used with equivalent concepts such as, variable, field, and other suitable concepts; and while the concept "data structure" is used for in the embodiments of the systems and methods described above, it will be evident that such concept can be interchangeably used with equivalent concepts such as, entity, object, class, and other suitable concepts.

Embodiments were chosen and described in order to best describe the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention, the various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalents.